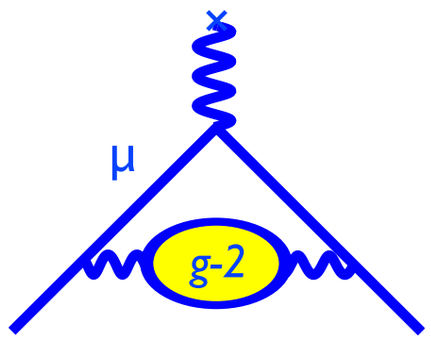


GM2-doc-2468

Debugging and Visualization for Sanity

Adam L. Lyon
FNAL/SCD

January 2015

1 INTRODUCTION

When writing scientific code, it is often useful to visualize some aspect of your program to check that everything makes sense. Such examples of “debugging for sanity” include,

- Ensuring that geometry looks right
- Are particles being produced in the right place?
- Do particles have the right momentum?
- More generally, is your algorithm doing the right thing?

Hopefully the sooner you debug your code for sanity, the more correct your code will be as you write it.

There are two problems with debugging for sanity:

1. Collecting the data to be visualized
2. Visualizing it with some data visualization application like Root, R, python, or ParaView

Collecting the data means determining what data you want to view and then somehow getting that out of your program. Typically, one opens a file, collects the data in some object that loops over events, and then at the end of the job write that data to the file and close the file. Or you write the data to a log file.

These options involve *altering* your code to perform this debugging task. You probably don’t want to run this code once you’ve past this point in development, so you may surround the debugging code with pre-compiler macros to turn it on and off. This makes your code ugly and hard to read. Or you’ll just delete this code, but you may need it again some day later.

This document describes how to use the OSX Xcode Debugger to *instrument* your code to collect the desired data *without altering* the code itself.

Debugging for sanity should also lead to writing useful unit tests that you should add to your codebase.

There are limitations to this method. Here we assume that sanity checks means running or creating a small sample of events for a quick check. If you need a large statistical sample (millions of data points), then you need to do something more substantial, like write a special Art analyzer to extract the desired data.

The debugger only has access to objects in memory. If you need to run a function to get the desired data, then this mechanism may not work.

Furthermore, for OSX you need to build the code you are debugging (you cannot easily debug libraries in the release). See GM2-doc-2459 (E989 Note 51) for more information on using your Mac for development and debugging.

We’ll follow an example of debugging the production of particles by the muon gas gun in the `gm2ringsim` simulation.

1.1 PREPARATIONS

We are going to debug for sanity the creation of particles by the Muon gas gun in `gm2ringsim`. For this example, we will need to build `gm2ringsim` ourselves (remember on the Mac, if you want to debug it, you must build it). We also need to build it from the *debug* release. Using the *prof* may work, but in many cases the variables you may want to examine will be optimized away or inaccessible. Remember, we are debugging for sanity so we will be making small data samples. Therefore, speed should not be a huge issue.

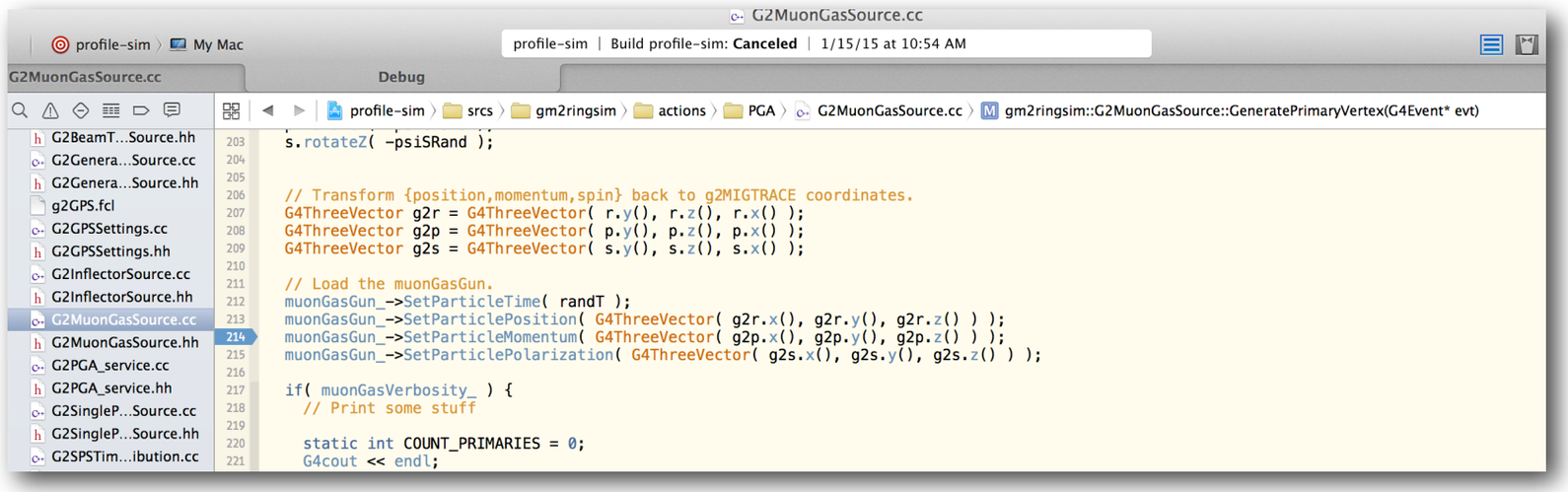
Set up a development area using the latest `debug` release that is available. You can find this out by doing

```
ups list -aK+ gm2 | grep debug
```

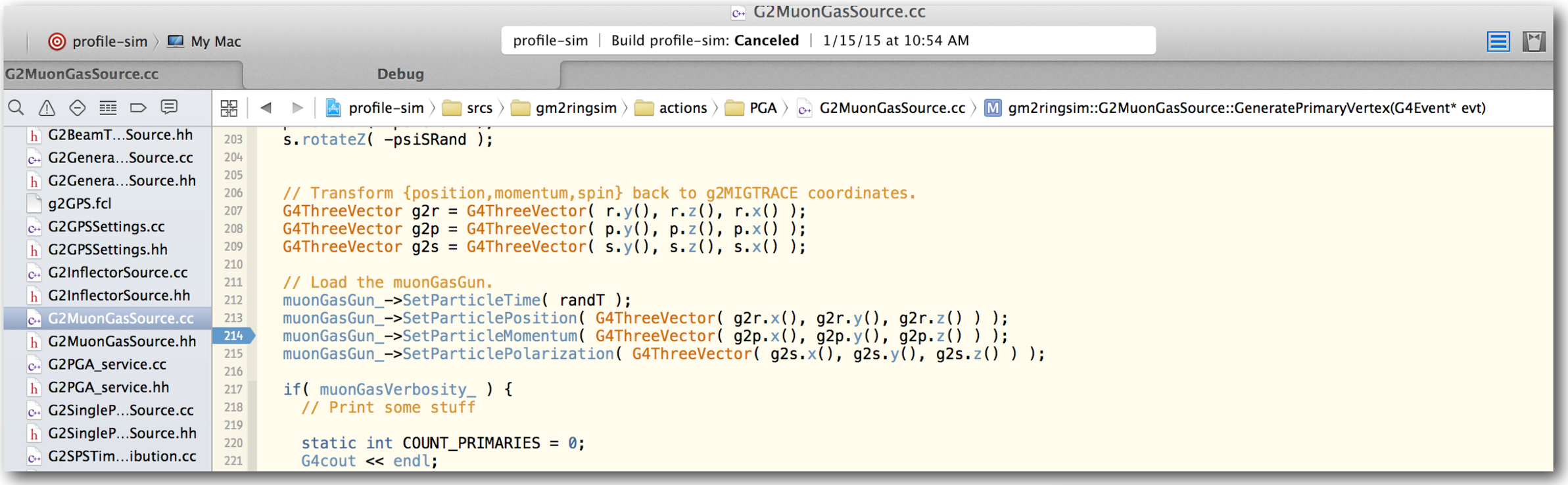
For this example. we'll be using `gm2 v5_00_00 -q e6:debug`. Since this is a base release, we need to build `artg4`, `gm2geom`, `gm2dataproducts` and `gm2ringsim` ourselves. If there was a point release, then perhaps we could have only built `gm2ringsim` ourselves, but there is no such debug point release when this article was written.

All of these topics are covered in the Offline Computing and Software Manual (GM2-doc-1825) [or in the release] and GM2-doc-2459 (E989 Note 51).

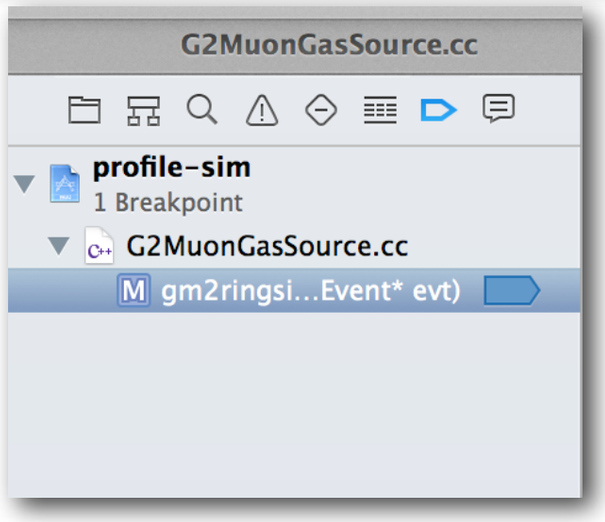
Looking at the code, we see that a good place to check sanity is in `gm2ringsim/actions/PGA/G2MuonGasSource.cc` line 214. That is where the generated particle's position and momentum are set. Let's examine these quantities.



2 EXPLORING THE CODE



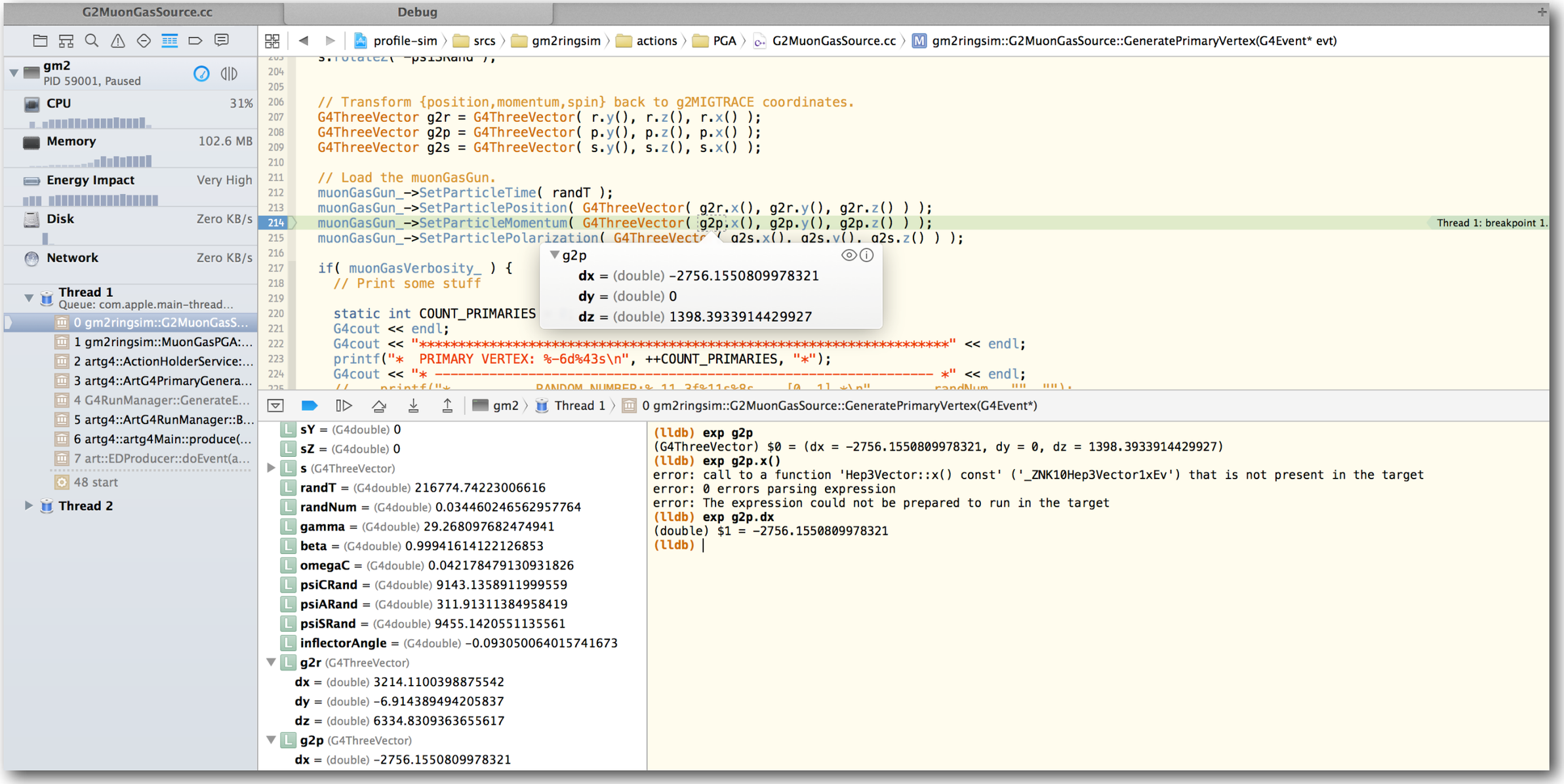
As shown in the example above, if you click on a line number in the source editor, you will create a breakpoint (e.g. line 214). Do that. If you switch to the Breakpoint navigator, the breakpoint will show up there as well, as seen on the right.



Now run the debugger executing the following command,

```
gm2 -n 200 -c ProductionMuPlusMuonGasGun.fcl
```

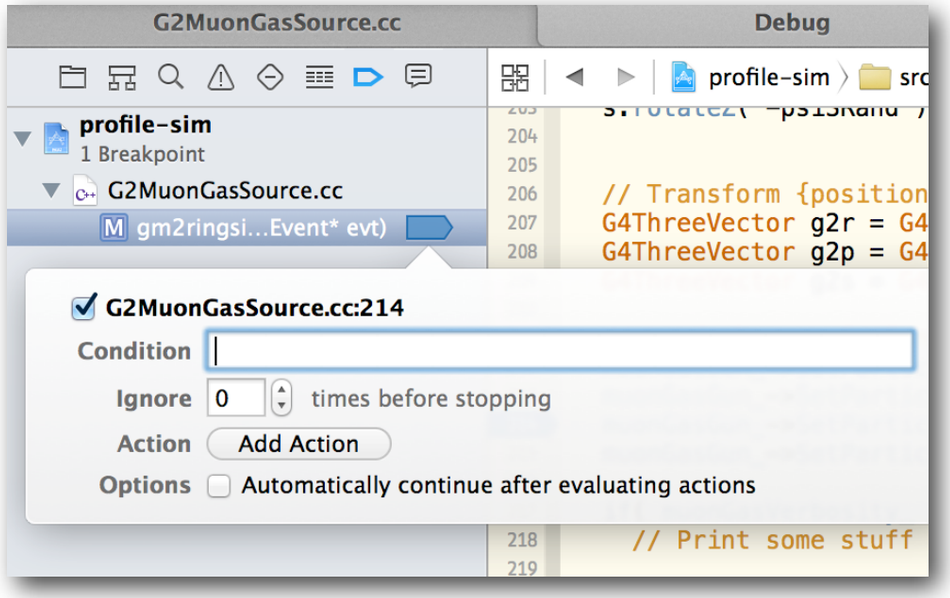
Since you are running a debug build, execution may be slower than what you are used to.



Above is an example of the debugging session. Execution has stopped at the breakpoint (line 214, noted by the green shaded bar). Let's look at what variables we have available. From the code, we see that `g2r` and `g2p` have the created muon's position and initial momentum respectively. These are `G4ThreeVector` objects. The debugger can see inside them, as shown in the variable viewer (lower left white background). You can also hover over the variable name (callout box in the code). We see that the object has three members. There are also functions, but we cannot call them from the debugger (lldb window lower right), because that code in `Hep3Vector` is not accessible. But that's ok. We can get what we want by using the `dx`, `dy`, `dz` components.

3 COLLECTING THE DATA

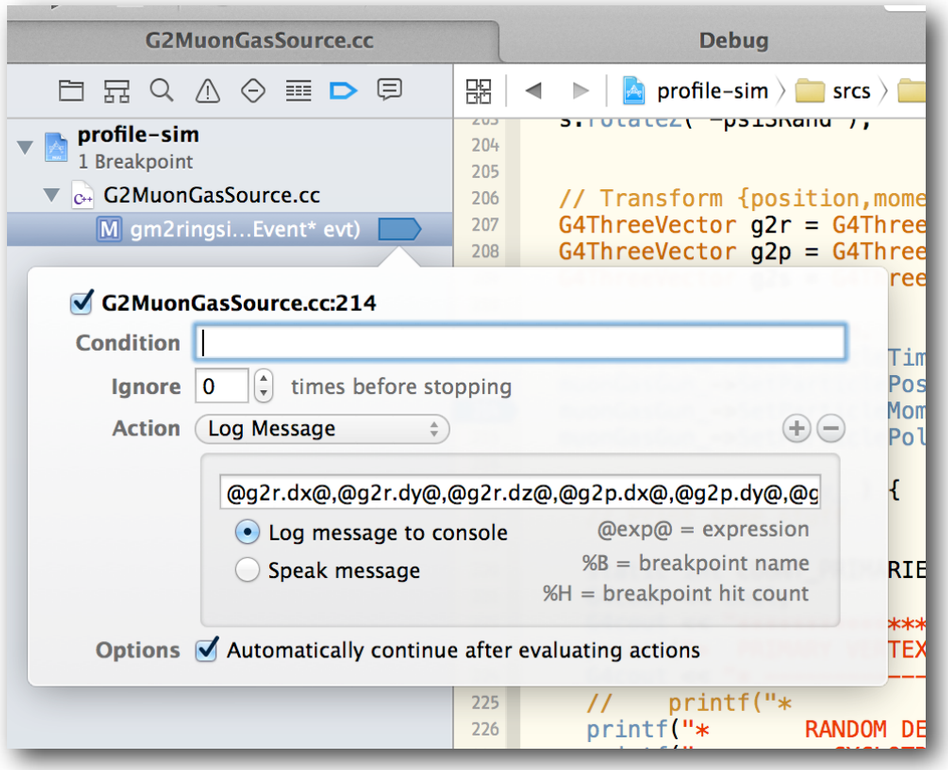
Let's check the sanity of the initial position and momentum of muons created by the gas gun. It looks like we want to examine the components of `g2r` and `g2p`. If the debugger is running, quit it (*Debug > Detach*). We'll now add this instrumentation.



Go back to the Breakpoint navigator, *right (control-) click* on the breakpoint and choose *Edit Breakpoint...* A screen will appear (figure at upper right of this page). Breakpoints have a lot of configuration options. Click on the *Add Action* button and choose *Log Message*. We can now write an LLDB expression to the log console. Let's write a comma-separated set of values (csv). Type in the text box,

```
@g2r.dx@,@g2r.dy@,@g2r.dz@,@g2p.dx@,@g2p.dy@,@g2p.dz@
```

and be sure the checkbox at the bottom is checked to continue after evaluating actions. That will make the debugger continue instead of stopping at the breakpoint. This configuration is shown in the figure on the lower right of this page. Note that each expression is surrounded by "@".



Now run the debugger again. If the debugger pane is not displayed, do *View > Debug Area > Show Debug Area*. As the code runs, you will see the console area of the debugger fill with our output! See figure on right.

Select all of the text (click inside and *Edit > Select All*) and copy (*Edit > Copy*).

Now we must get this information into a file. Go to your terminal window and open a new file with emacs,

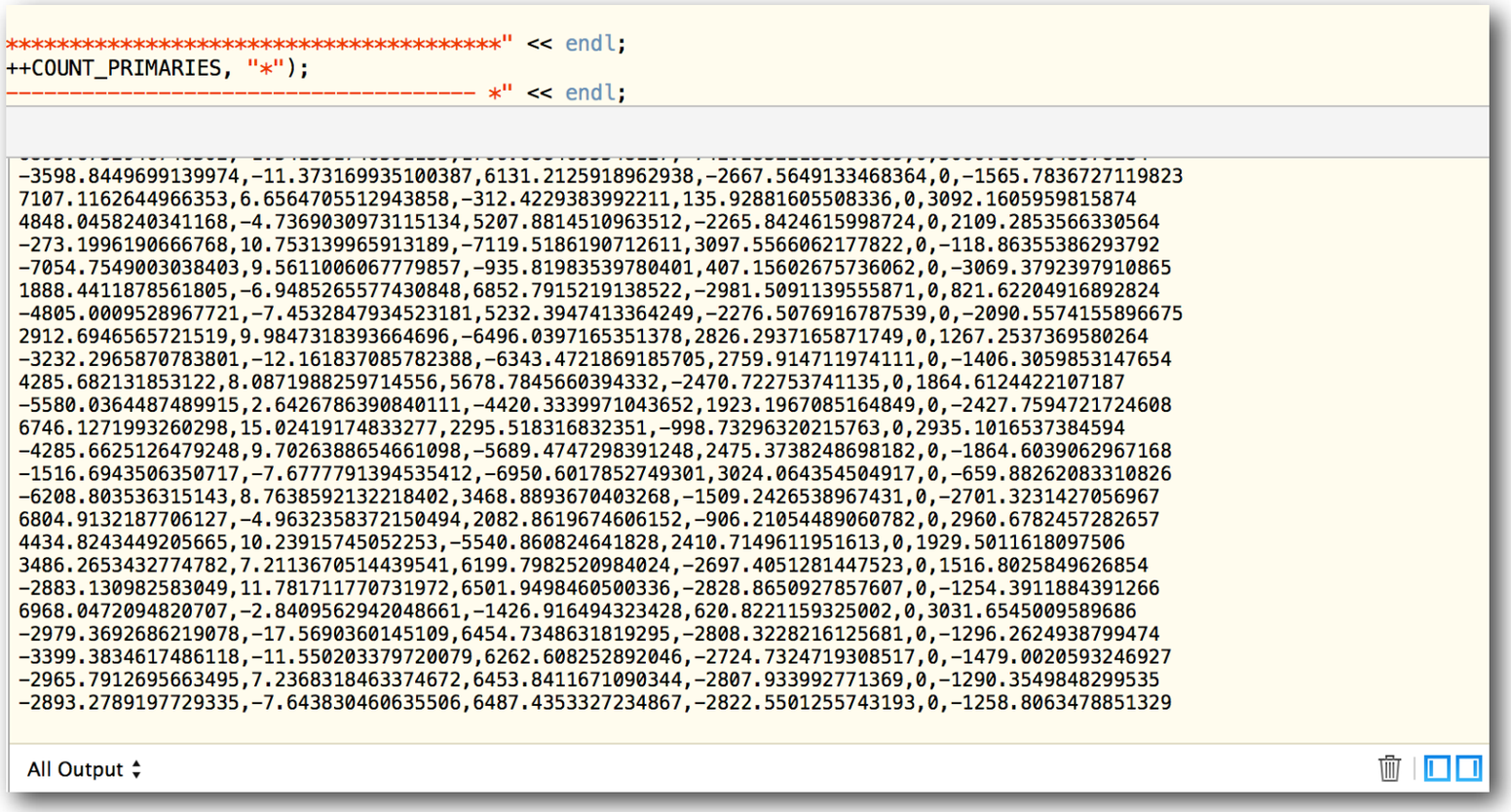
```
emacs -nw data.csv
```

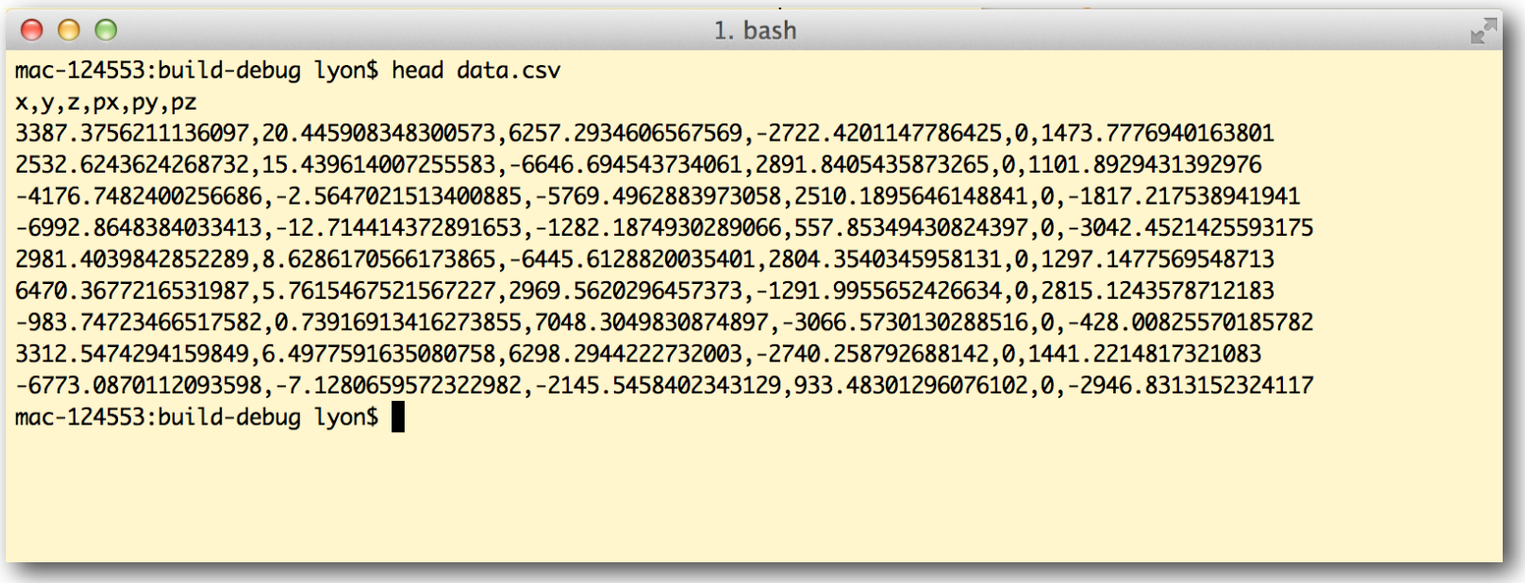
On the first line, let's type the header

```
x,y,z,px,py.pz
```

and press *enter* for a new line. Now paste in the data with ⌘*V*. Save the file with *Control-X Control-C* to quit and then *y* to save.

If you do `head data.csv`, you should see something like the figure on the lower right.





In this example, we wrote the desired data to the debugger console. You can also configure the breakpoint to feed this data into a shell or python script for processing and/or storing.

In the Breakpoint navigator, *right (control-) click* on the breakpoint and choose *Edit Breakpoint...* Then click on the *Action* chooser and choose *Shell command*. You can also press the + and add multiple actions to the breakpoint. You must give the full path to your script and the arguments list (each argument separated by a comma; so the log message we did earlier works here too).

Make sure the script is executable and starts with the appropriate `#!`, or else Xcode will give errors.

An example script is given on the right. It first checks if the output file exists, and if it does not, it writes the header line. Otherwise it outputs the data to the file.

Such scripts are more convenient than cutting and pasting.

```
#!/bin/bash

out="data.csv"

if [ ! -r "$out" ]; then
  echo 'x,y,z,px,py,pz' > $out
fi

echo $1,$2,$3,$4,$5,$6 >> $out
```

3.1 SAVING THE BREAKPOINTS

So long as you don't delete the breakpoints, they will remain in your Xcode project. You can make a breakpoint inactive and Xcode will ignore it.

There is no built-in way to reconstitute a breakpoint if the Xcode project is lost (or, say, someone else wants to try the breakpoint). Since breakpoints are easy to create, a best practice would be to have a *debugging* directory in the product source with any scripts that were used for data collection along with a text file describing the breakpoints. The needed information would be

- File and line number of the breakpoint. It may be useful to have a copy of the nearby source code in case the line number changes due to changes in the source.
- Information about actions, including log information, script name and arguments, etc.
- A description of the output data

4 VISUALIZATION

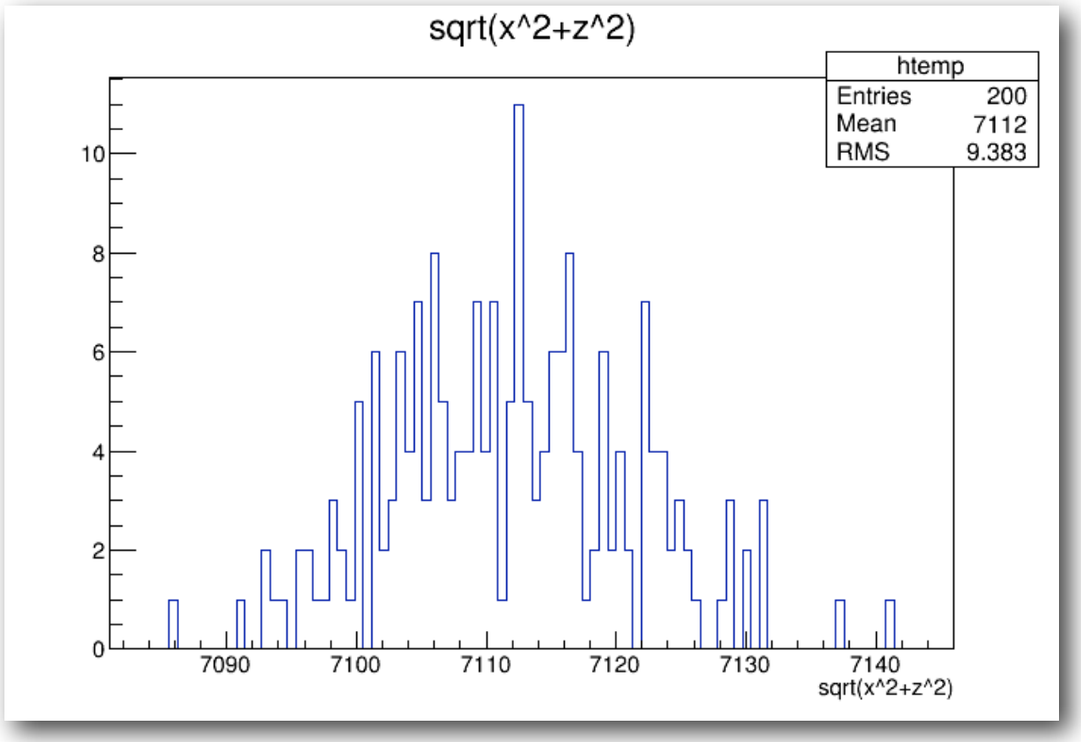
Now that we have our data, we should look at it. There are many avenues to explore here. We will try three,

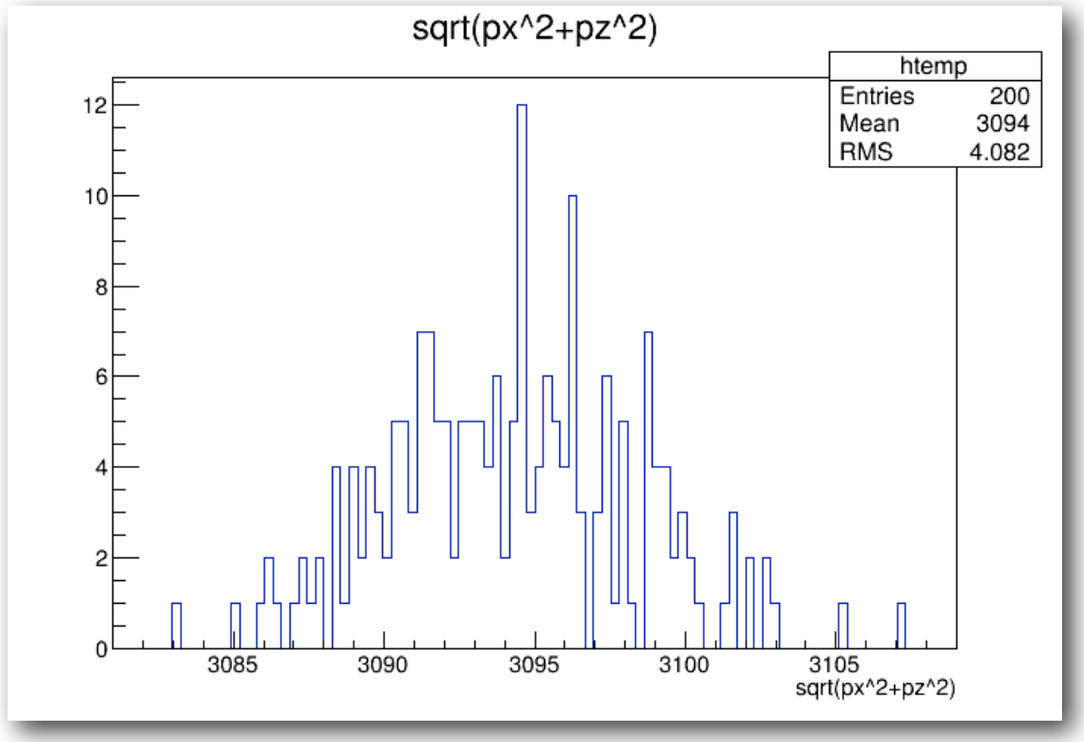
- Plots in *Root*
- Plots in *R*
- 3D visualization in *ParaView*

4.1 ROOT

Loading the data into *Root* is possible with `TTree::ReadFile`. You'll then have the data in a Root tree and you can use all of the tools for visualization and analysis that Root provides. See transcript below. Note that Root does not like the header line and complains, but fortunately it just skips it. Two example plots are at bottom: the radial position and momentum around p_{magic}.

```
root [0] TTree *T = new TTree("ntuple", "muon creation data")
root [1] Long64_t nlines = T->ReadFile("data.csv", "x:y:z:px:py:pz", ',')
Warning in <TTree::ReadStream>: Couldn't read formatted data in "x" for branch x on line 1; ignoring line
Warning in <TTree::ReadStream>: Read too few columns (1 < 6) in line 1; ignoring line
root [2] T->GetEntries()
(const Long64_t)200
root [3] T->Draw("sqrt(x^2+z^2)")
root [4] T->Draw("sqrt(px^2+pz^2)")
```





4.2 R

R an open source data analysis application (see *http://www.r-project.org/*) that is well suited for analyzing simple data files. The R libraries of *dplyr* and *magrittr* make for a very interesting and easy to read and write data analysis workflow (once you understand how things work). If you are interested in *R*, be sure to try out the *RStudio* application.

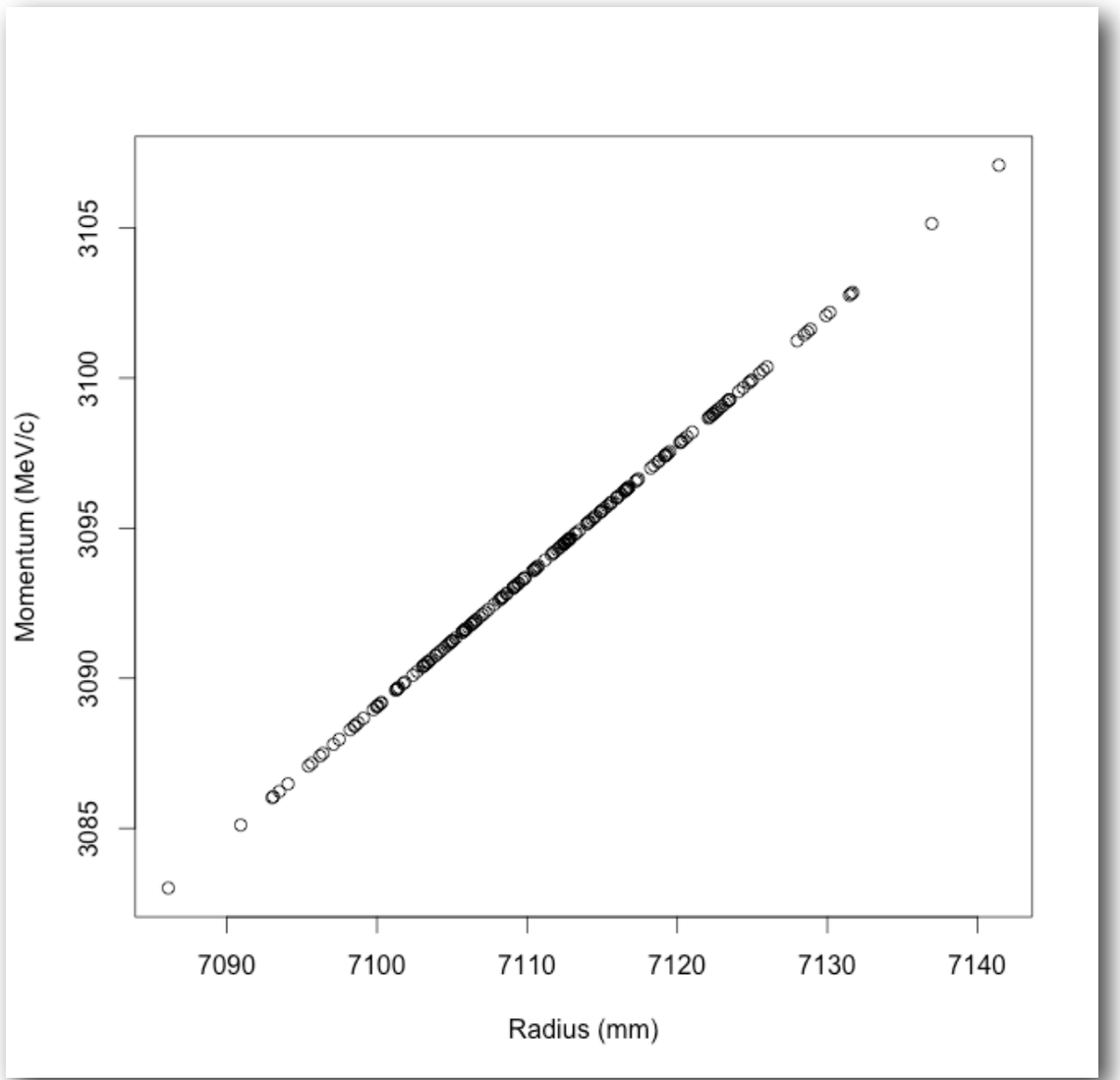
See the example code and plot to the right. We see evidence of a correlation put into the code.

```
require(dplyr) ; require(magrittr)

d = read.csv("data.csv", header=T) %>% tbl_df

# Add r and p columns
d = d %>% mutate( r = sqrt(x^2+z^2), p = sqrt(px^2 + pz^2) )

# Plot them
d %$% plot(r, p, xlab='Radius (mm)', ylab='Momentum (MeV/c)')
```



4.3 PARAVIEW

ParaView is an extremely sophisticated 2D/3D visualization application and framework used for analysis of large scale simulations (e.g. air flow over the space shuttle). See www.paraview.org for more information and downloading instructions. For this example we will use version 4.2.0.

As of writing this document, we do not have a geometry description that works well in *ParaView*. That may be coming. In the meantime, we'll make do with what we have.

Following our example, we have position and momentum information for each created particle. We can plot this information in 3D space. To visualize, we will create a little arrow (a *glyph*) to indicate the direction of the momentum vector.

Note that *ParaView* is a big and complicated program. We will only scratch the surface here. There are many sites on the web with additional information.

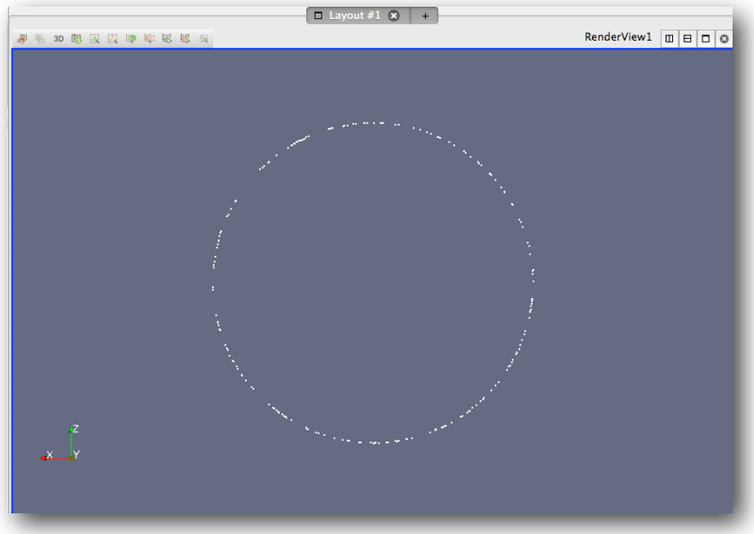
After launching *ParaView*, load in the `data.csv` file with the menu selection *File > Open...* Unfortunately, ParaView does not use the standard Mac file open dialog, so you will have to hunt around to find the file.

A common theme with ParaView is that any potential expensive operation (like loading a file) does not complete until the user presses the *Apply* button. Press that now.

A spreadsheet should appear with our csv data. We need to do a few things before we can visualize the data fully.

We need to turn the data table into graphics points. Use the "Table to Points" filter. From the menu, select *Filters > Alphabetical > Table to Points*. In the properties box, select *x* for the X column, *y* for the Y column, and *z* for the Z column. And press the *Apply* button.

If you click on the little eye symbol next to *TableToPoints1* so that it is black, and then close the spreadsheet view (most likely layout 2). with the x, you will see Layout 1 and you should see dots forming a circle. These dots are the created muons. If instead you see a line, you have the wrong orientation. Click on the –Y axis button in one of the various toolbars. That will change the view to looking down onto the ring.



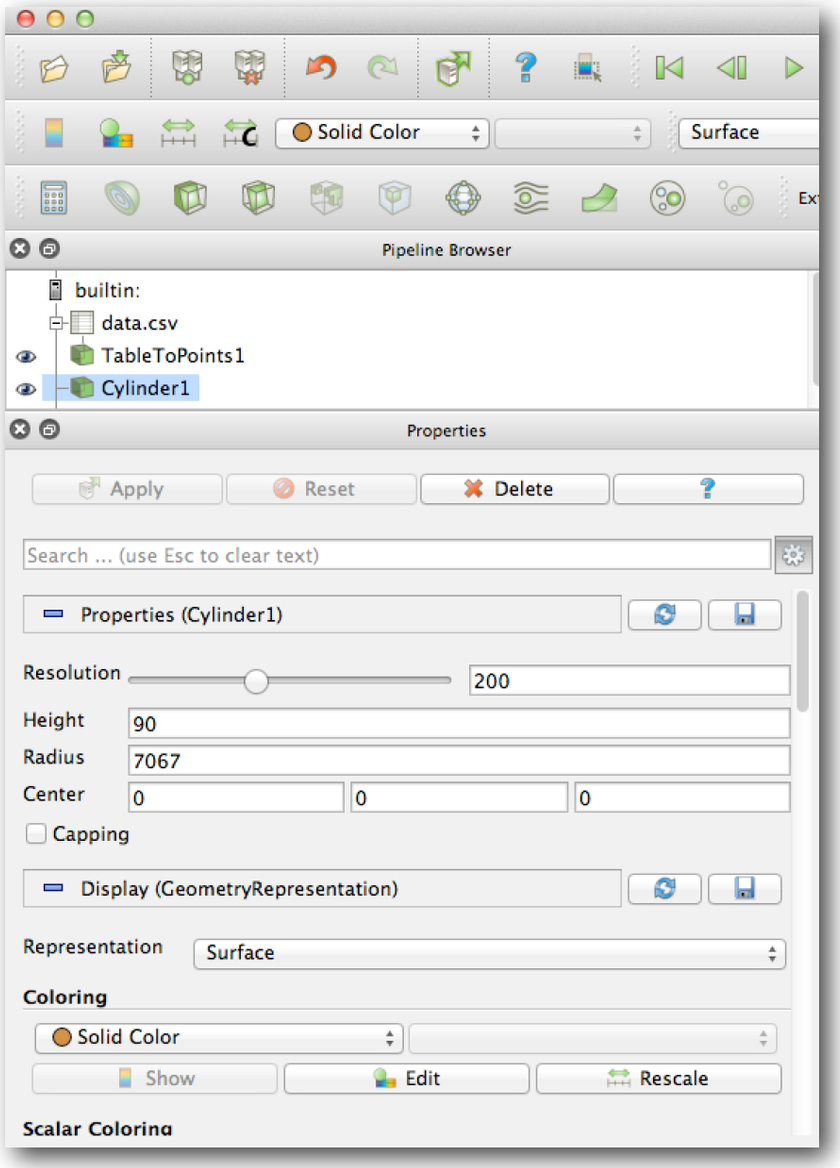
You can spin around the 3D view with your mouse.

We do not have any geometry on this image. Let's see if we can set up boundaries at the vacuum chamber edges. The magic radius is 7112 mm and the inner

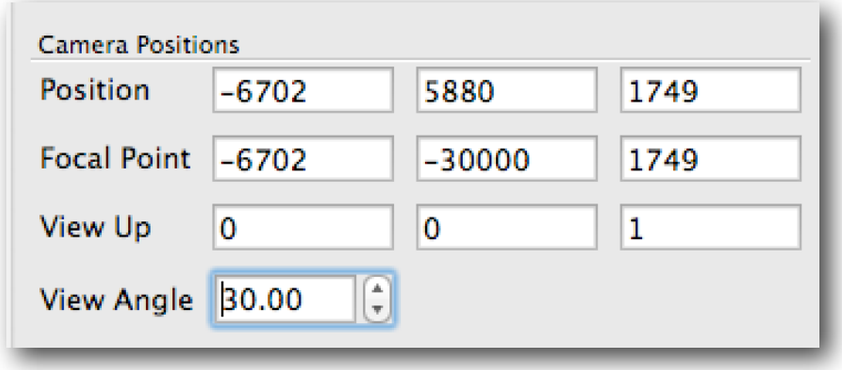
and outer walls are ±45 mm from there. The height of the storage region is ±45 mm as well. So let's create two uncapped cylinders, both centered at 0,0,0; both with a height of 90 mm. One will have a radius of 7067 mm and the other will have a radius of 7157 mm.

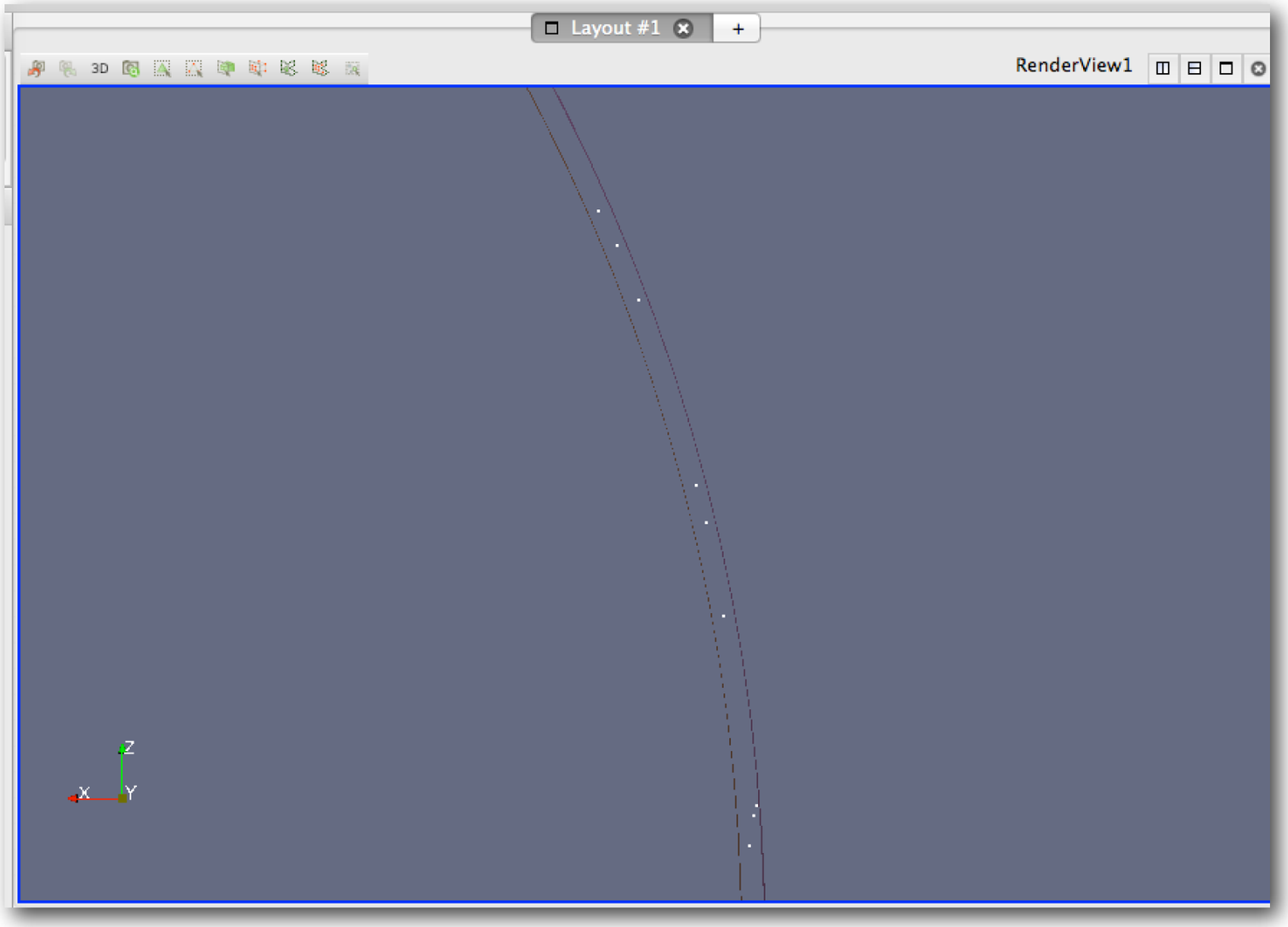
Make sure the *TableToPoints1* item is selected in the Pipeline viewer. Create a cylinder source from the menu by choosing *Sources > Cylinder*. Fill out the properties sheet as shown on the upper right and select *Apply*.

Now create another Cylinder, make it the same as the first one except the radius is 7157.



You can get a good view of the muons in the ring by adjusting the camera (click on the little camera icon in the image window, next to "3D") and fill in the positions as shown on the lower right.





Here is the view from doing the previous actions. The dots are the muons in the ring. They all look inside.

Now, let's add the the momentum directions. First, we have to turn the momentum components into a vector. Make sure the *TableToPoints1* item is selected in the pipeline viewer. From the menu, choose *Filters > Alphabetical > Calculator*. For *Result Array Name*, enter `pvec`. In the text box below, enter `px*iHat+py*jHat+pz*kHat` and then press *Apply*. Let's add another calculator determine the momentum magnitude. Add another calculator filter. For *Result Array Name*, enter `pmag`. In the text box below, enter `mag(pvec)` and then press *Apply*.

Now with *Calculator2* selected in the pipeline, choose from the menu *Filters > Alphabetical > Glyph*. Then enter (leave items not mentioned at their defaults),

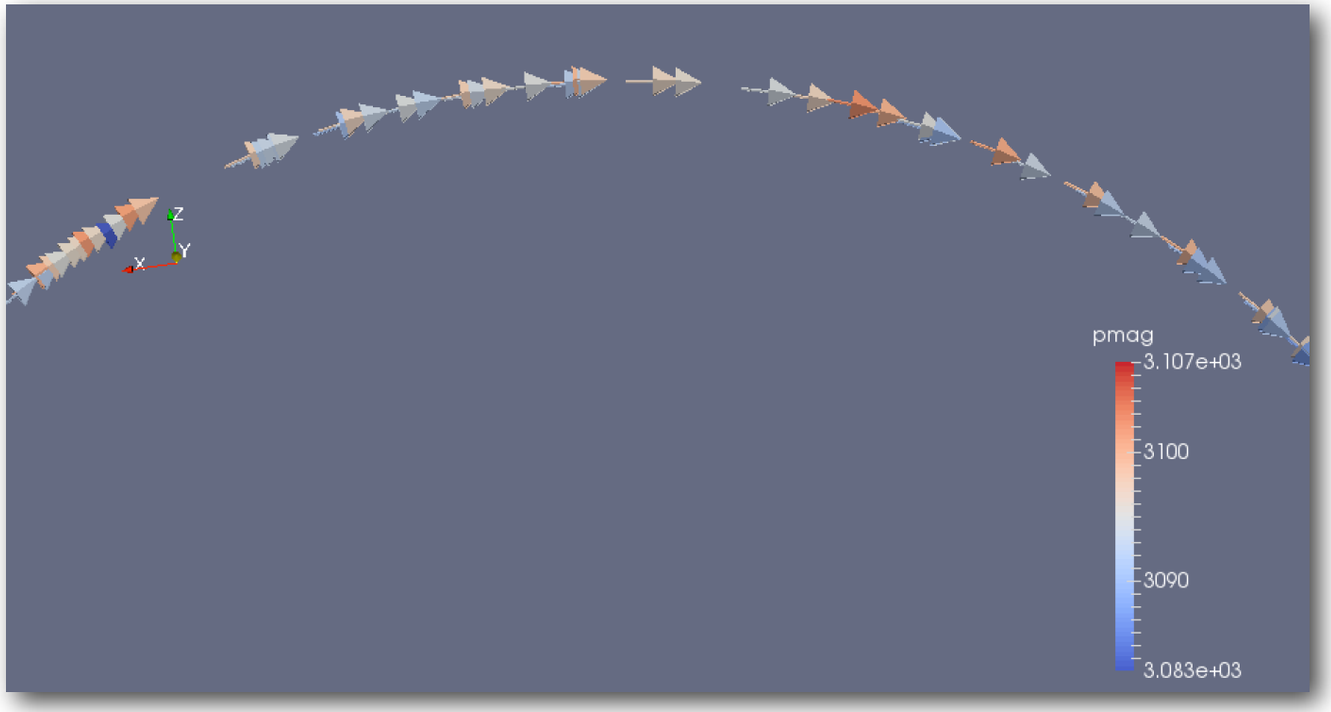
```
Tip Resolution = 20
Tip Radius = 0.3
Tip Length =  0.5
Scalars = pmag
Vectors = pvec
Scale Mode = off
Scale Factor = 400
Glyph Mode = all points
```

then press *Apply*.

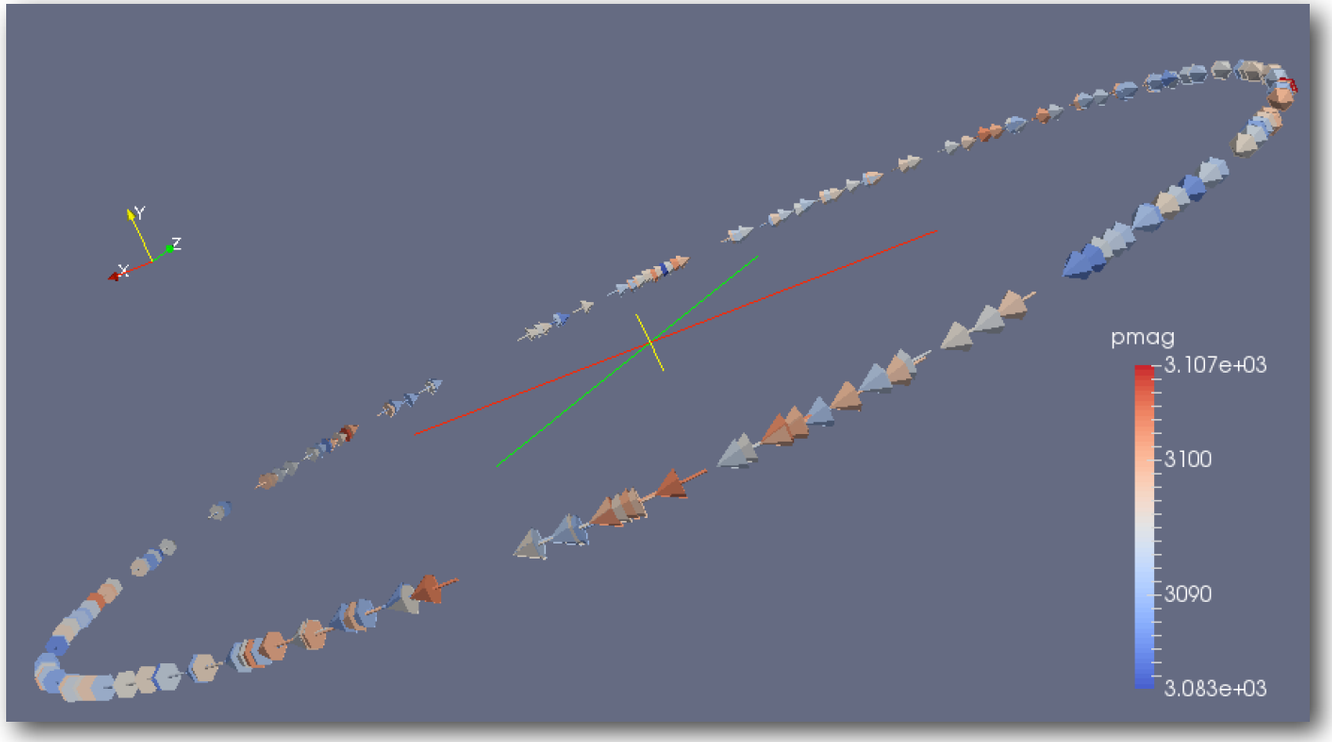
In the pipeline viewer, find the TableTo-Points1 item and click on the eye icon to make it grey, hiding the points from view.

We see that the momentum vectors (the arrows) are tangential to the ring. The color is the momentum magnitude.

The cylinders have been hidden for a better view of the muons.



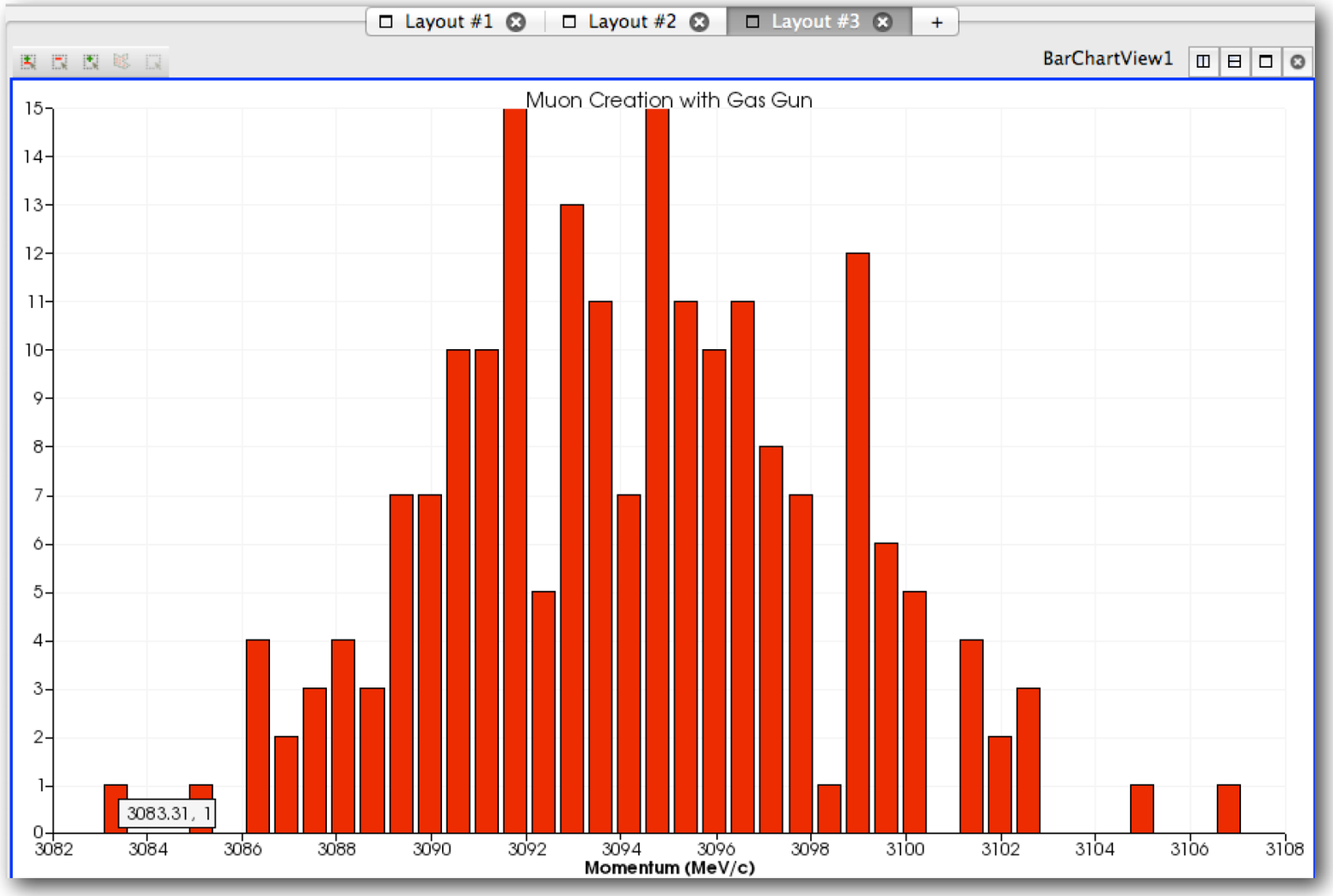
Looking down from above



Looking from a side angle

ParaView can do 2D plots as well.

We've only barely scratched the surface of what this program can do.



5 CONCLUSIONS

Writing code is hard. Writing code correctly is even harder. The ability to check what a program is doing is essential to having it work the right way. Debugging is an important part of a developers toolkit, but doesn't lend itself to collecting many data points and visualizing the results.

This note documents debugging and visualization for sanity - a way to collect data from your program and display it without altering your code. The Xcode debugger makes it quite easy to inspect and then extract data to a file. The data may be viewed with a variety of applications. One hopes by viewing the data, one can confirm that the program is operating correctly, or quickly see errors and fix them before they become entrenched in the code.